Einführung in die Informatik

Jochen Hoenicke

Software Engineering
Albert-Ludwigs-University Freiburg

Sommersemester 2014

Teil VIII

Programmieren und Datenstrukturen

Algorithmen

Algorithmus

Ein Algorithmus ist eine Vorschrift zur Lösung eines Problems.
Folgende Eigenschaften sind essentiell

- Finitheit: Die Vorschrift (also das Programm) muss endlich sein, darf zu jedem Zeitpunkt nur endlich viel Speicher benutzen,
- Terminiertheit: Ein Algorithmus darf nur endlich viele Schritte insgesamt durchführen.
- Effektivität: Für jede Anweisung eines Algorithmus muss klar sein, was sie macht und wie man sie ausführt.

Weitere wünschenswerte Eigenschaften:

- Determiniertheit: Das Ergebnis ist immer das selbe.
  Determinismus: Der Rechenweg ist fest vorgegeben.

Beispiele

- Ein Kochrezept ist finit. Es erfüllt Terminiertheit. Die Effektivität hängt von der Vorbildung des Kochs/der Genauigkeit der Beschreibung ab. Determiniertheit ist oft nicht gegeben (... schmecken Sie mit Salz und Pfeffer ab).
- Algorithmus zum Multiplizieren von mehrstelligen Dezimalzahlen: Finitheit, Terminiertheit, Effektivität und Determiniertheit sind gegeben. Determinismus nicht unbedingt.
  Tatsächlich ist der Begriff eine Abwandlung des Namens des Gelehrten Muhammed al-Chwarizmi, der in seinem Lehrbuch das „Über das Rechnen mit indischen Ziffern“ (um 825) solche Algorithmen vorgestellt hat.

Euklidischer Algorithmus

Einer der ältesten Algorithmen stammt von Euklid (3. Jahrhundert v.u.Z)

Die Elemente des Euklid, Buch VII, Proposition 2

Es soll der größte gemeinsame Teiler von AB und CD, die nicht teilerfremd sind bestimmt werden.
[...] Wenn CD nicht Teiler von AB ist, subtrahiert man, von den beiden Zahlen AB und CD ausgehend, immer die kleinere von der größeren bis die entstandene Zahl Teiler der ihr vorhergehenden ist, der dann der größte gemeinsame Teile von AB und CD ist.

Der Rest der Proposition gibt ein Beispiel und beweist die Korrektheit des Algorithmus.

Euklidischer Algorithmus in Java

Man kann diesen Algorithmus direkt als Java-Programm implementieren:

```
public int computeGCD(int ab, int cd) {
    if (ab <= 0 || cd <= 0)
        throw new IllegalArgumentException();
    while (ab % cd != 0) {
        if (ab > cd) {
            int new = ab - cd;
            ab = cd;
            cd = new;
        } else
            cd -= ab;
    }
    return cd;
}
```

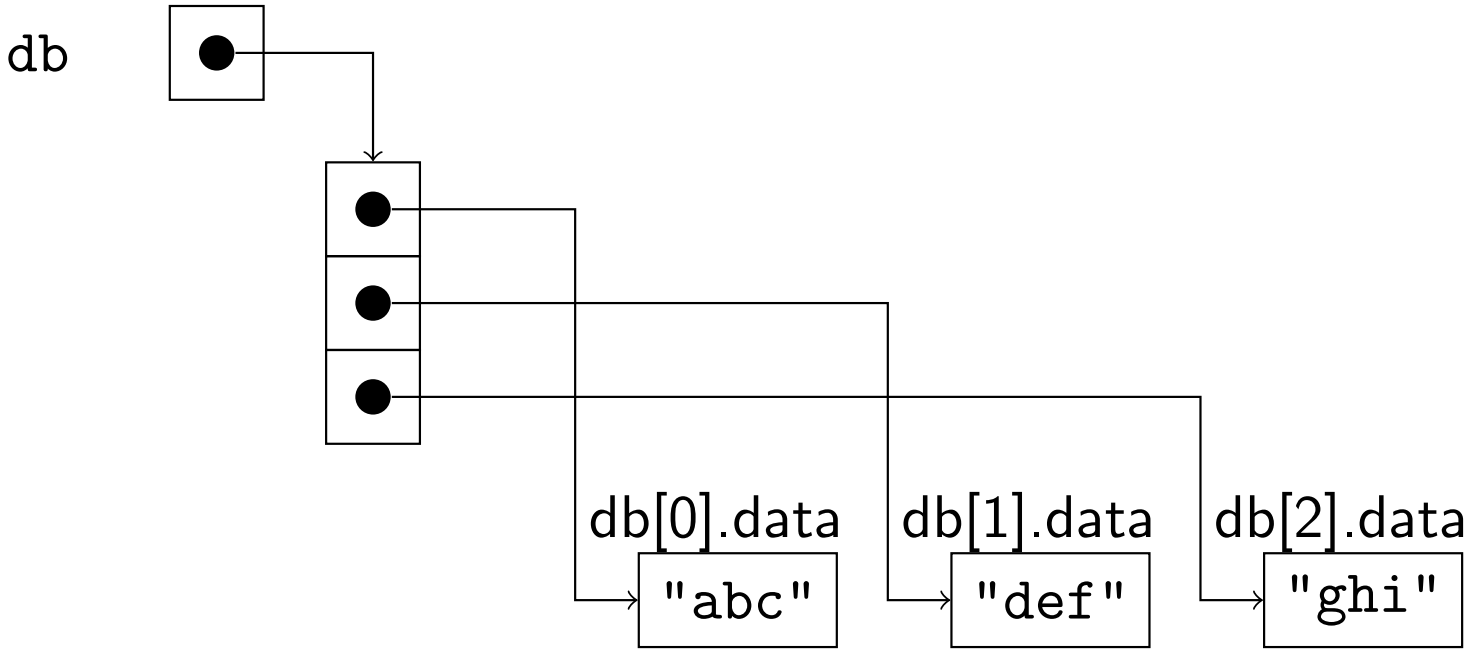
Euklidischer Algorithmus in Java

Leicht verbesserte Variante:

```
public int computeGCD(int ab, int cd) {
    if (ab < 0 || cd < 0)
        throw new IllegalArgumentException();
    while (cd != 0) {
        int new = ab % cd;
        ab = cd;
        cd = new;
    }
    return ab;
}
```

Verkettete Listen

Objekte in Java werden als Referenzen (Zeiger) gespeichert.



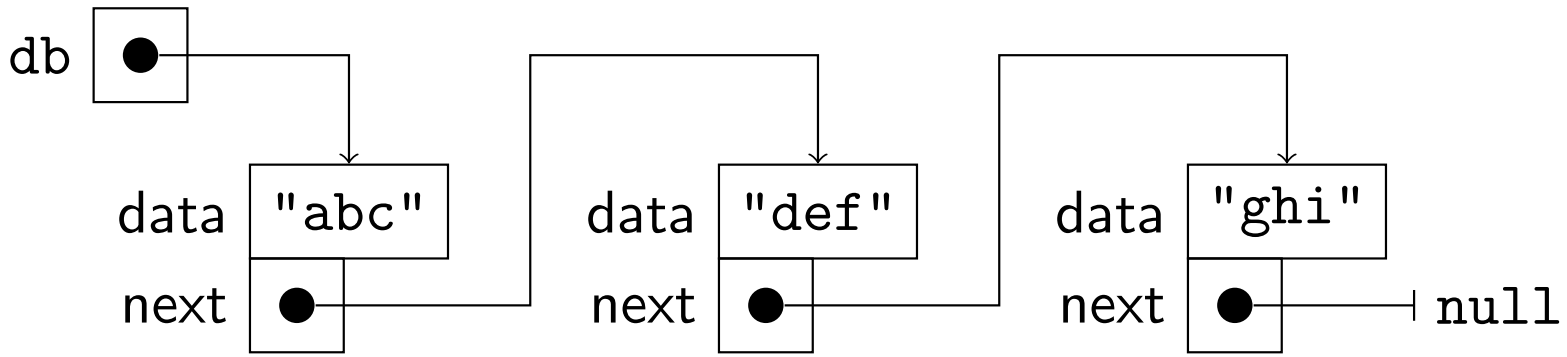
Eine weitere Möglichkeit mehrere Objekte zu speichern, ist sie zu verketten.

Verkettete Liste

```
class ListEntry {
    Object    data;
    ListEntry next;
}
```

Verkettete Liste

Eine Komponente im Objekt ist eine Referenz auf das nächste Objekt.



Einfach verkette Listen

Das Beispiel auf der letzten Folie zeigte eine einfach verkettete Liste. Eine Implementierung ist:

```
public class List
    ListEntry head;

    /** Erzeugt eine neue leere Liste. */
    public List() {
      head = null;
    }

    /** Prueft, ob die Liste leer ist. */
    public boolean isEmpty() {
      return head == null;
    }
    ...
```

Einfach verkette Listen (2)

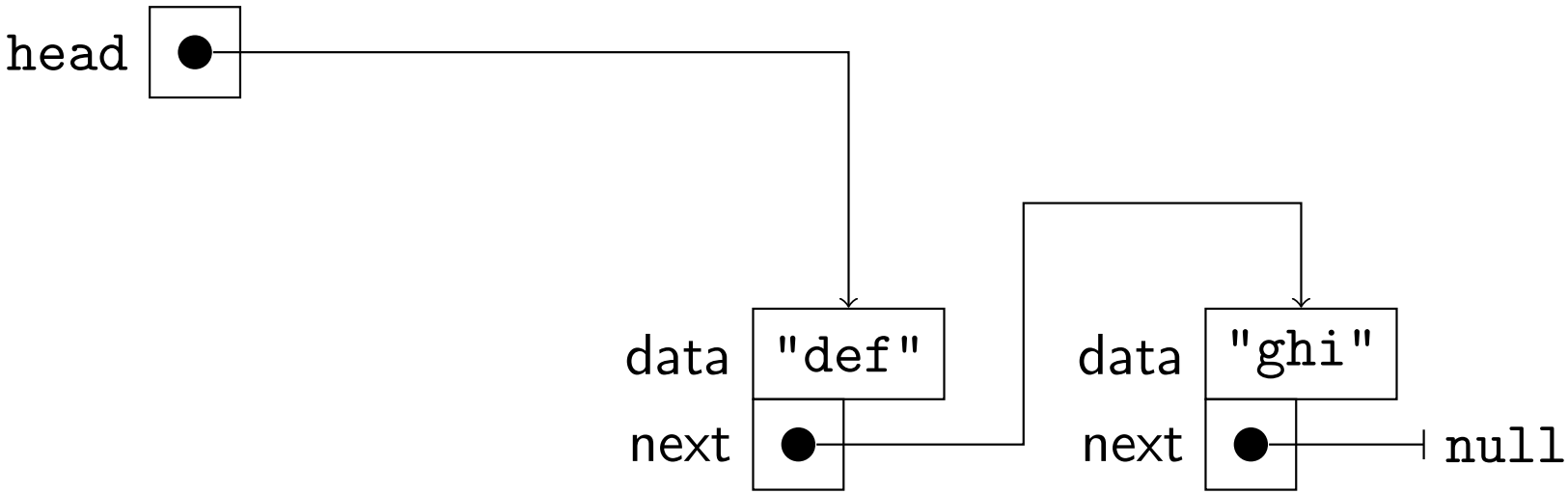
Um ein Element in eine Liste einzufügen, kann man es leicht als erstes Element einfügen.

```
//public class List
    ...
    /** Fuegt ein neues Element ein. */
    public void add(Object data) {
        ListEntry newEntry = new ListEntry();
        newEntry.data = data;
        newEntry.next = head;
        head = newEntry;
    }
    ...
```

Man kann aber auch an einer anderen beliebigen Stelle einfügen (siehe Übung).

Einfügen in Listen

Wir wollen `"abc"` in die Liste `["def","ghi"]` einfügen.

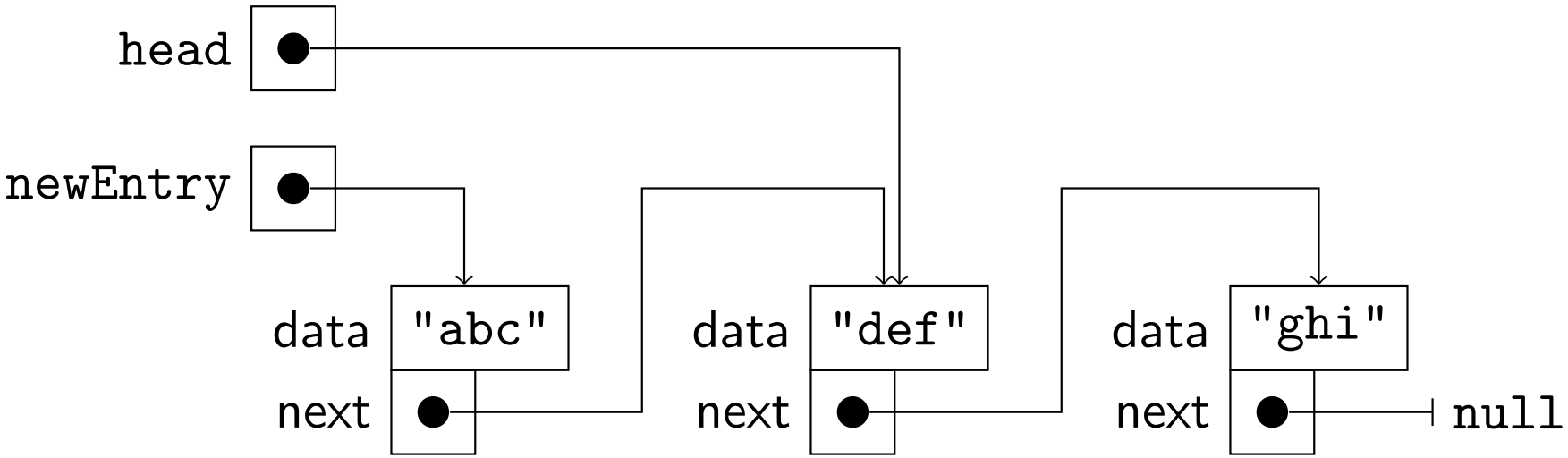


Einfügen in Listen

Wir wollen "abc" in die Liste ["def","ghi"] einfügen.

```
ListEntry newEntry = new ListEntry();
newEntry.data = "abc";
newEntry.next = head;
```

Die ersten drei Zeilen erzeugen einen neuen Listeneintrag



Einfügen in Listen

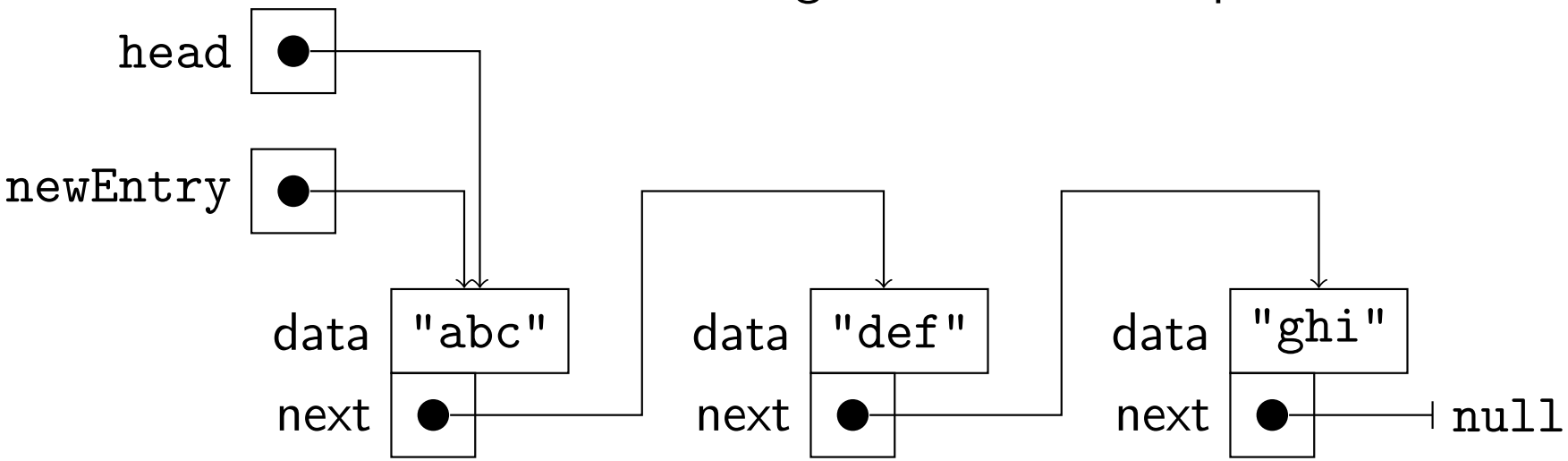
Wir wollen "abc" in die Liste ["def","ghi"] einfügen.

```
ListEntry newEntry = new ListEntry();
newEntry.data = "abc";
newEntry.next = head;
```

Die ersten drei Zeilen erzeugen einen neuen Listeneintrag

```
head = newEntry;
```

Die letzte Zeile macht diesen Eintrag zum neuen Startpunkt.



Iterieren

Um über eine Liste zu laufen, kann man sich durch die `next`-Referenzen hangeln:

```
//public class List
...
/** Gibt alle Elemente als String zurueck */
public String toString() {
    String content="";
    String comma="";
    for (ListEntry e = head;
         e != null; e = e.next){
        content += comma + e.toString();
        comma = ",";
    }
    return "[" + content + "]";
}
```

Löschen

Um ein Element zu löschen, muss es zunächst gefunden werden. Daher bauen wir folgenden Algorithmus:

1. Suche zu löschendes Element in der Liste. Merke dabei den Vorgänger.
2. Entferne das Element aus der Liste.

Löschen – Teil 1

Zunächst durchlaufen wir die Liste und merken uns dabei den Vorgänger.

```
//public class List
...
/** Entfernt das Element data aus der Liste. */
public boolean remove(Object data) {
    ListEntry prev = null;
    ListEntry elem = head;
    while (elem != null
           && !elem.data.equals(data)) {
        prev = elem;
        elem = elem.next;
    }  ...
```

Ist `elem == null`, so wurde das Element nicht gefunden. Andernfalls zeigt `elem` auf das zu löschende Element, `prev` auf den Vorgänger oder `null`, wenn das erste Element gelöscht werden soll.

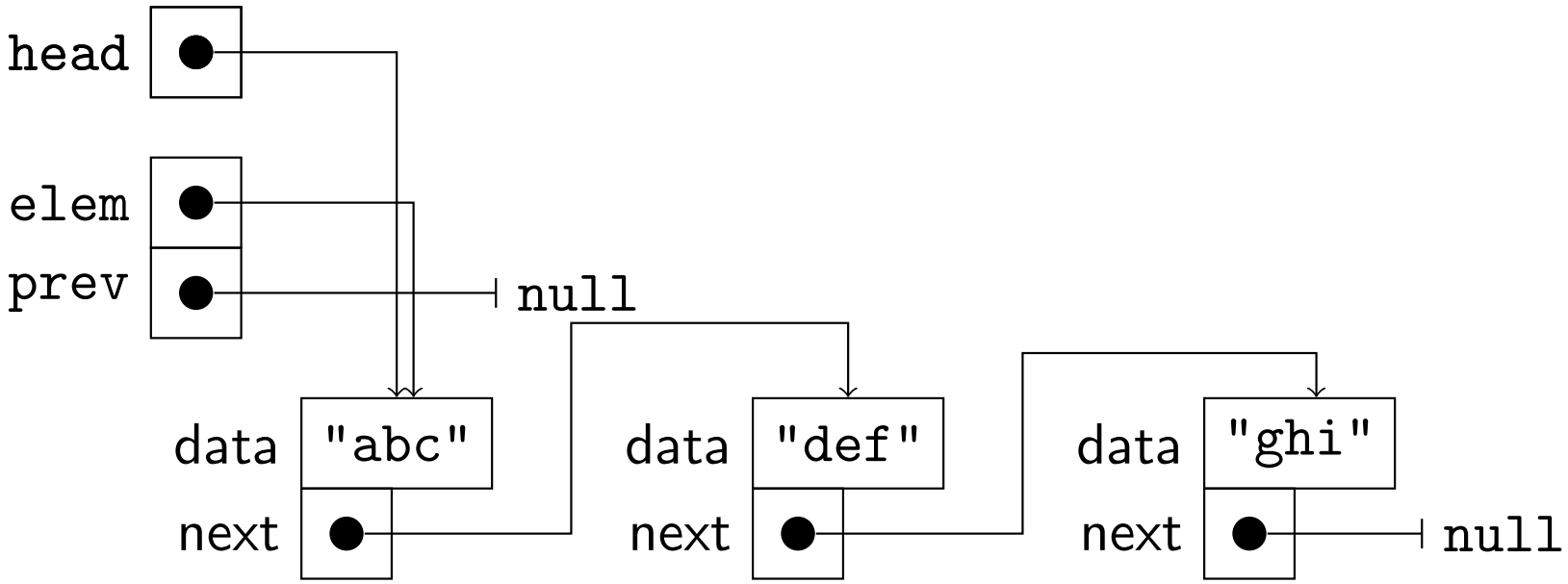
Löschen – Teil 2

```
...//boolean remove(Object data)
    if (elem == null) {
        // data wurde nicht gefunden.
        return false;
    } else if (prev == null) {
        // erstes Element soll geloescht werden.
        assert head == elem;
        head = elem.next;
        return true;
    } else {
        // haenge elem aus der Liste aus.
        assert prev.next == elem;
        prev.next = elem.next;
        return true;
    }
}
```

Löschen – Animation

Zunächst löschen wir `"def"`:

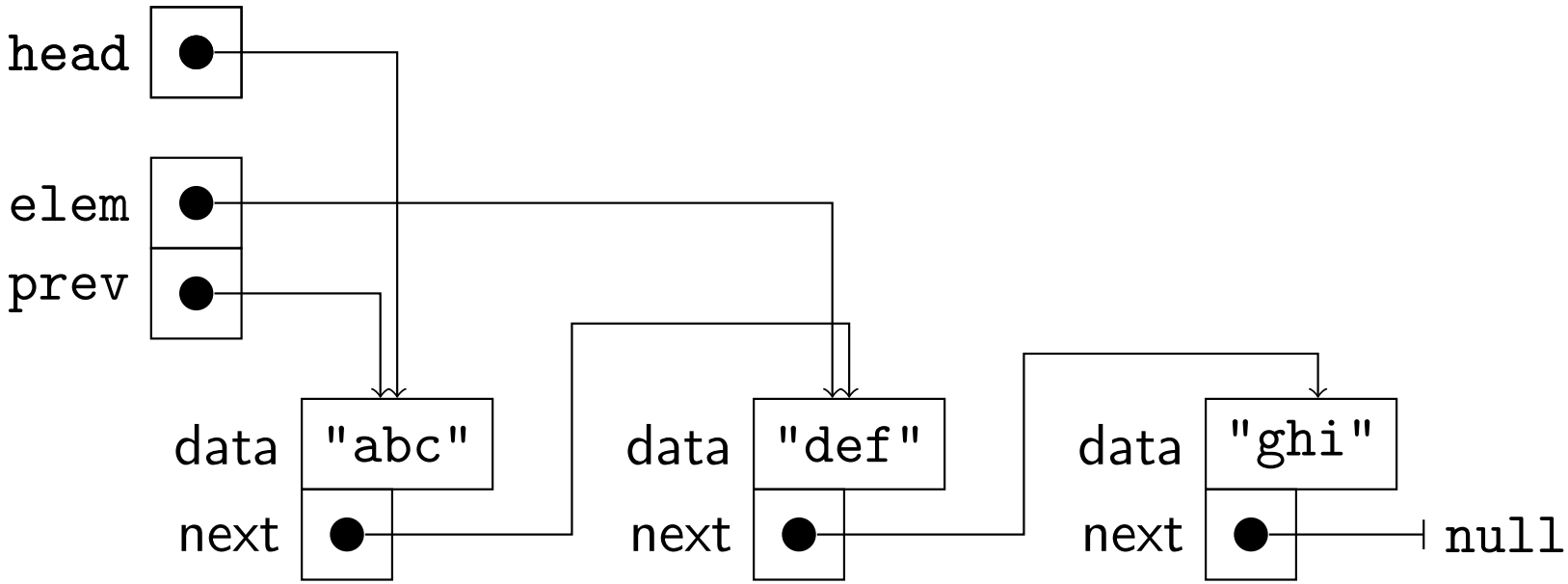
Initialisierung



Löschen – Animation

Zunächst löschen wir `"def"`:

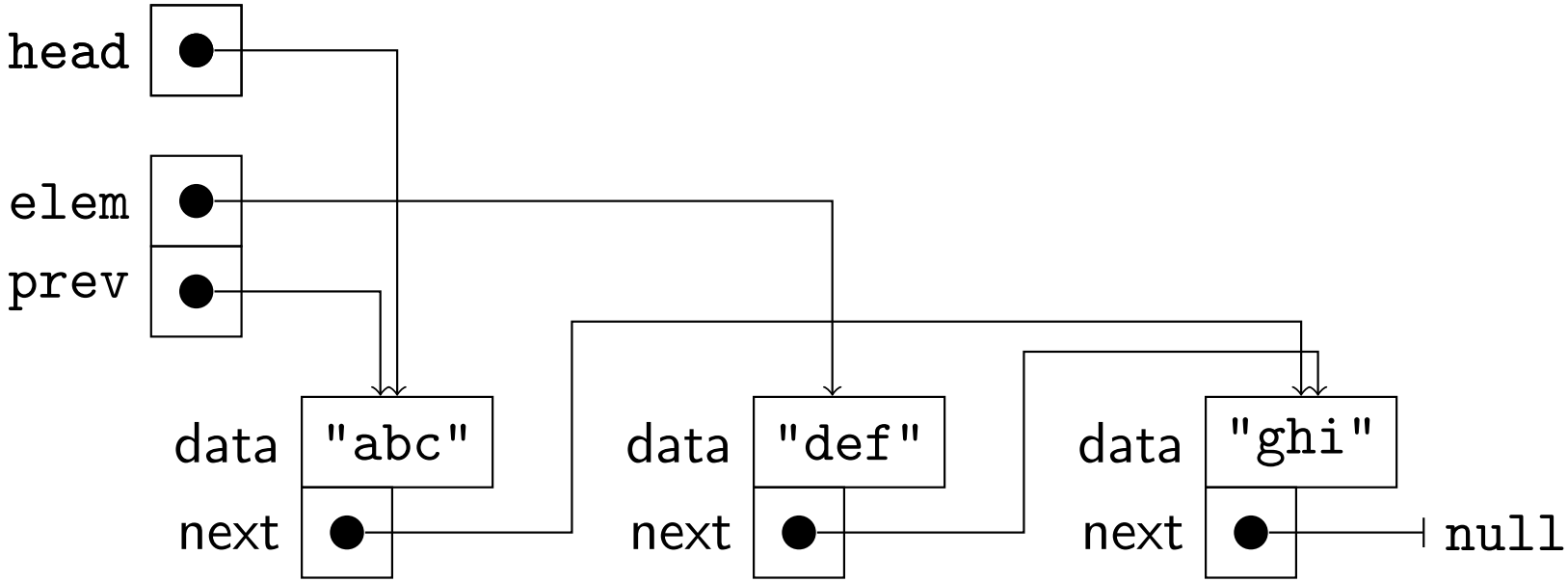
Suche `elem` mit `elem.data="def"`.



Löschen – Animation

Zunächst löschen wir `"def"`:

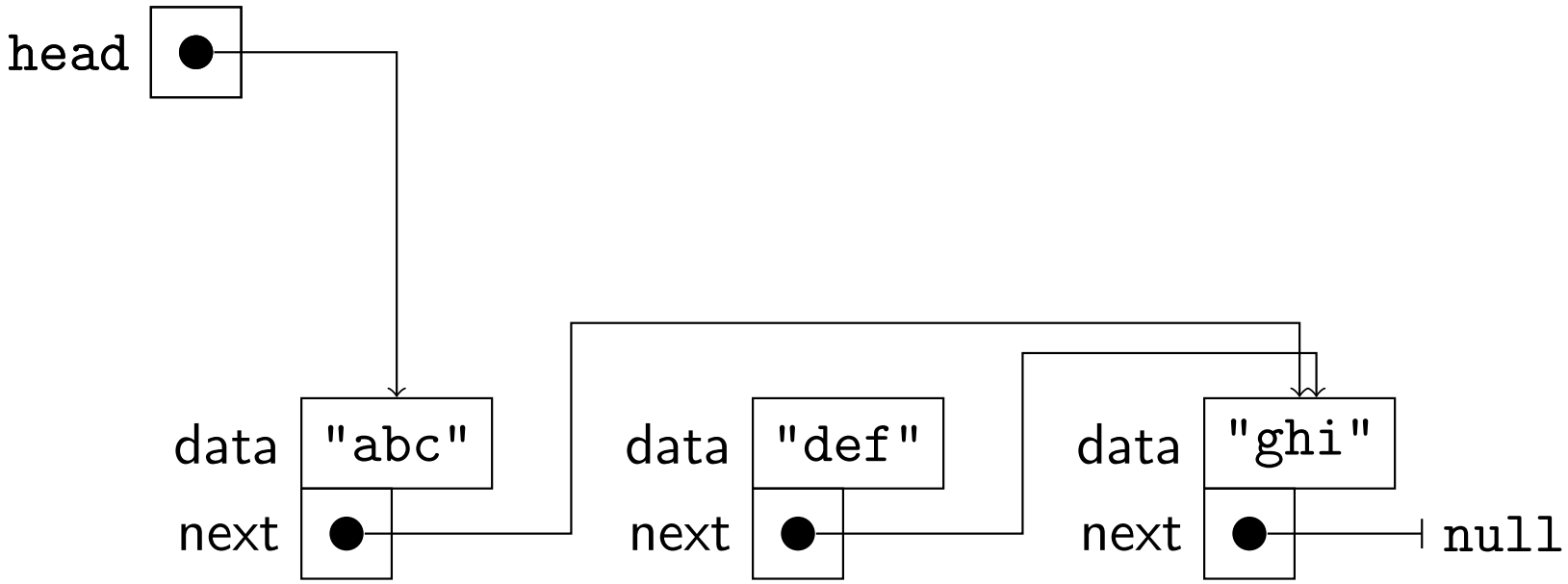
Aushängen (else-Teil).



Löschen – Animation

Zunächst löschen wir `"def"`:

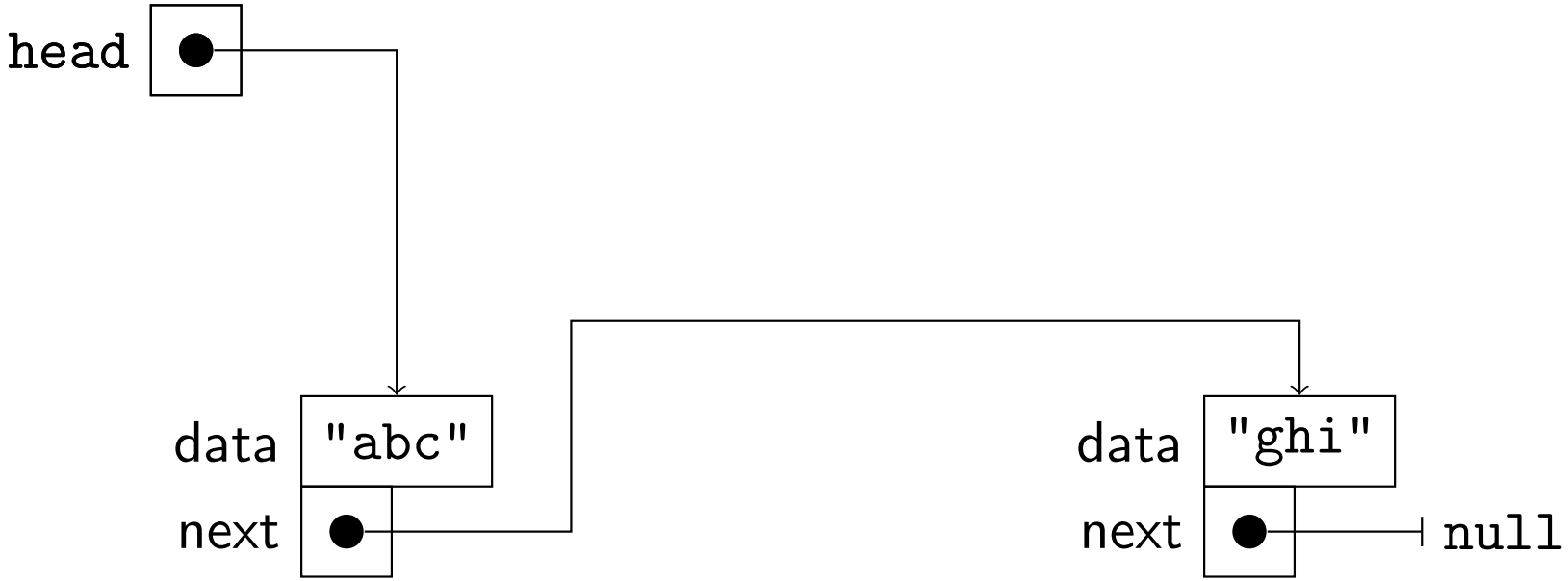
Methode gibt `true` zurück.



Löschen – Animation

Zunächst löschen wir `"def"`:

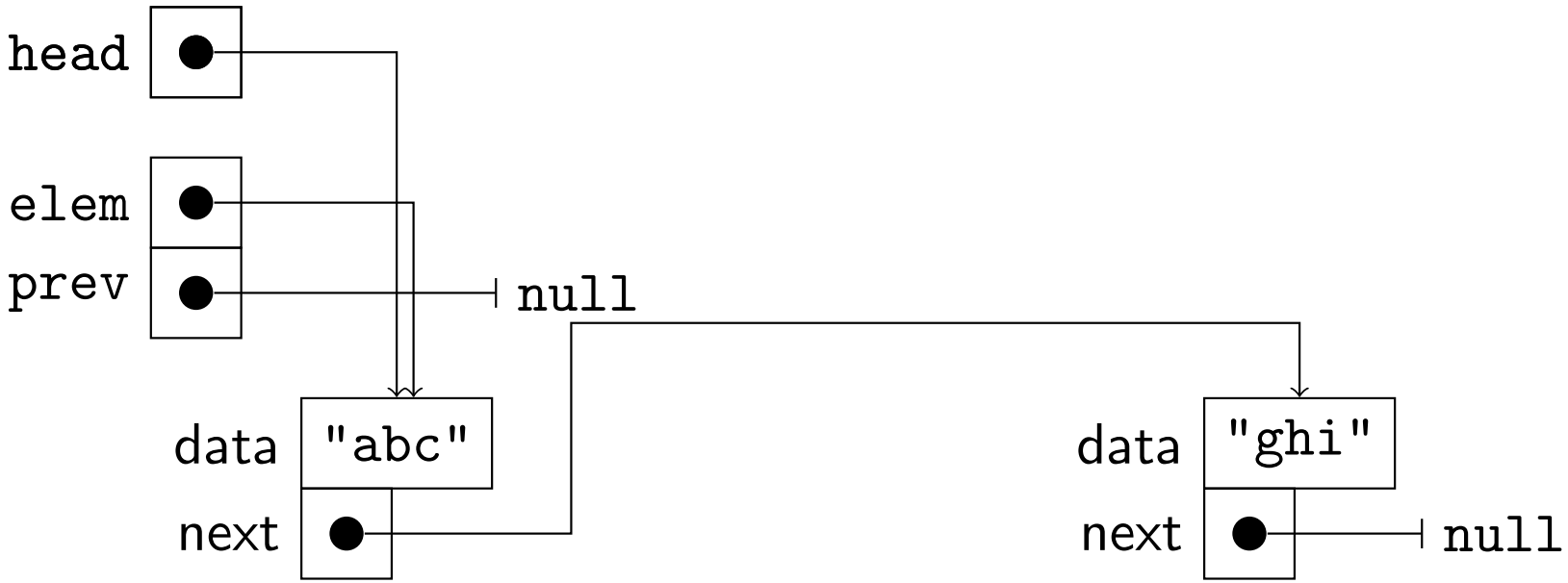
Müllsammlung (garbage collection) läuft.



Löschen – Animation

Jetzt löschen wir "abc":

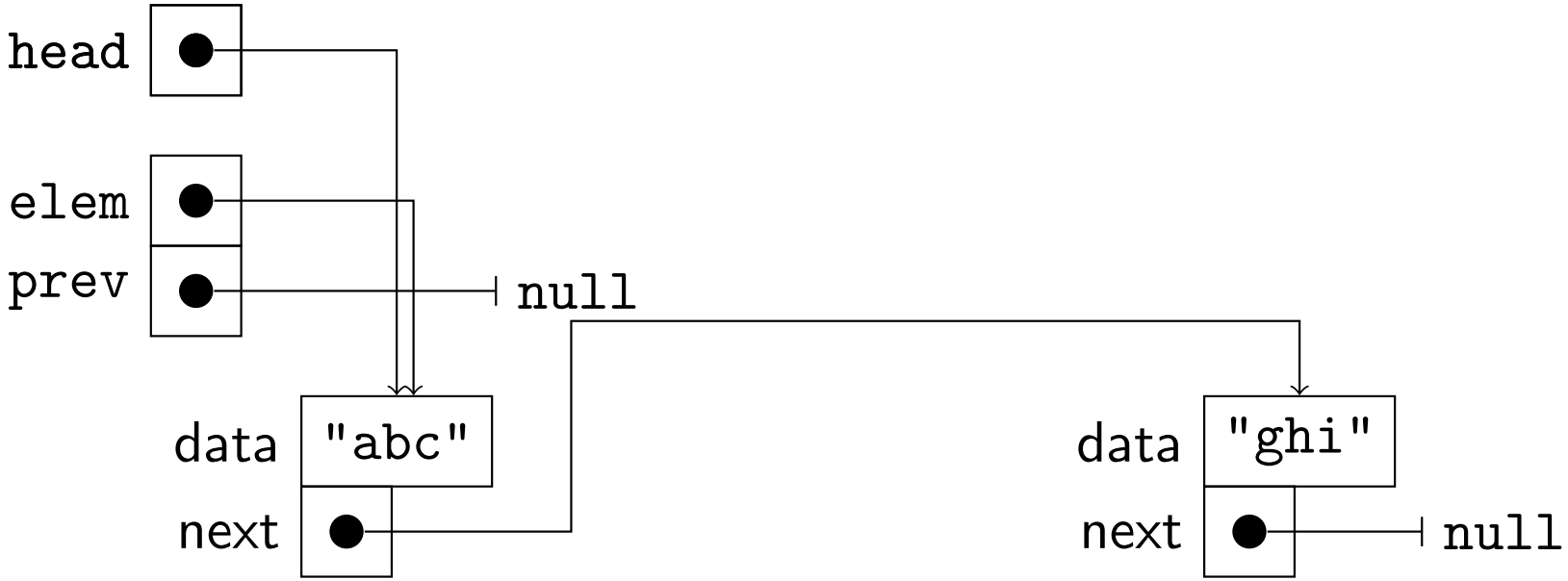
Initialisierung



Löschen – Animation

Jetzt löschen wir `"abc"`:

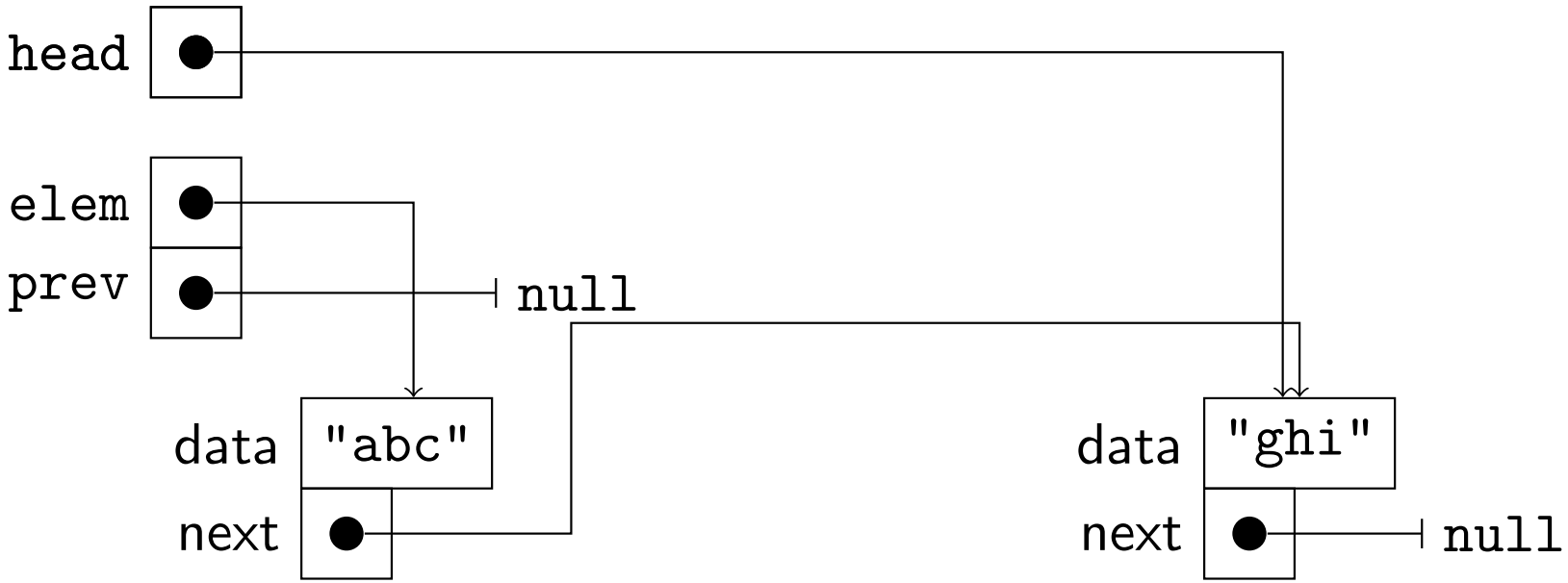
Suche `elem` mit `elem.data="abc"`.



Löschen – Animation

Jetzt löschen wir `"abc"`:

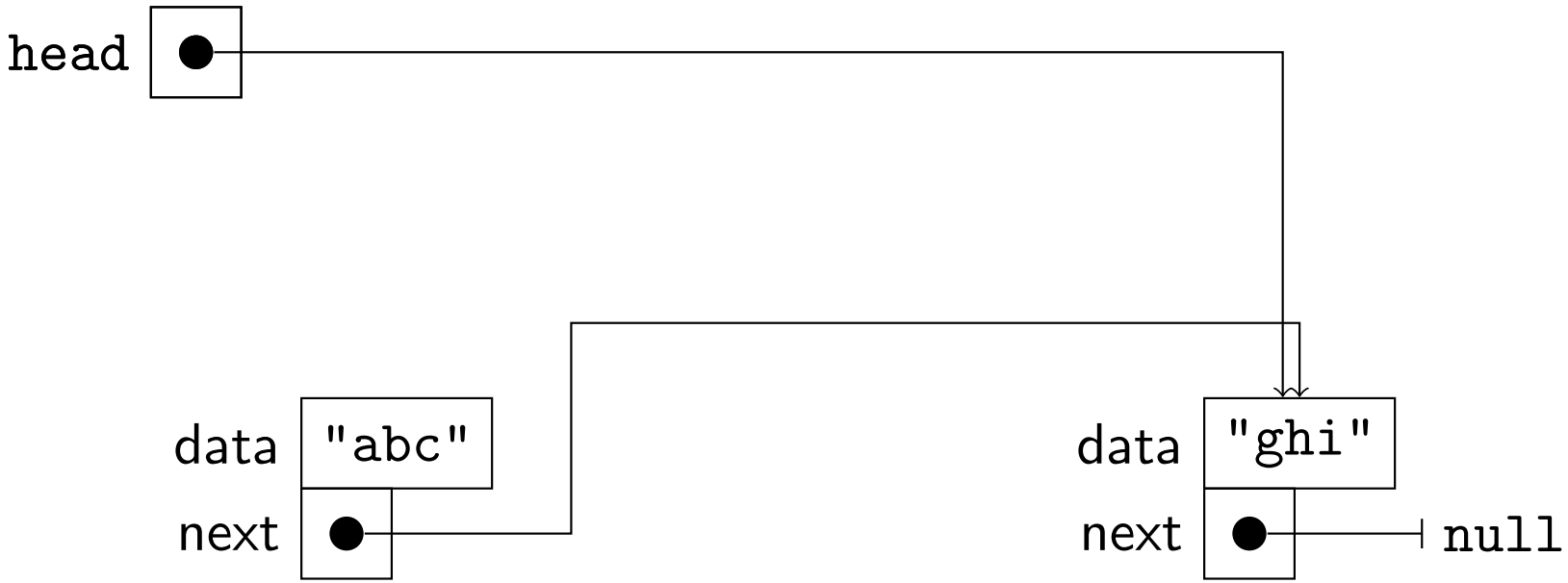
Aushängen (`prev == null`).



Löschen – Animation

Jetzt löschen wir `"abc"`:

Methode gibt `true` zurück.



Löschen – Animation

Jetzt löschen wir `"abc"`:

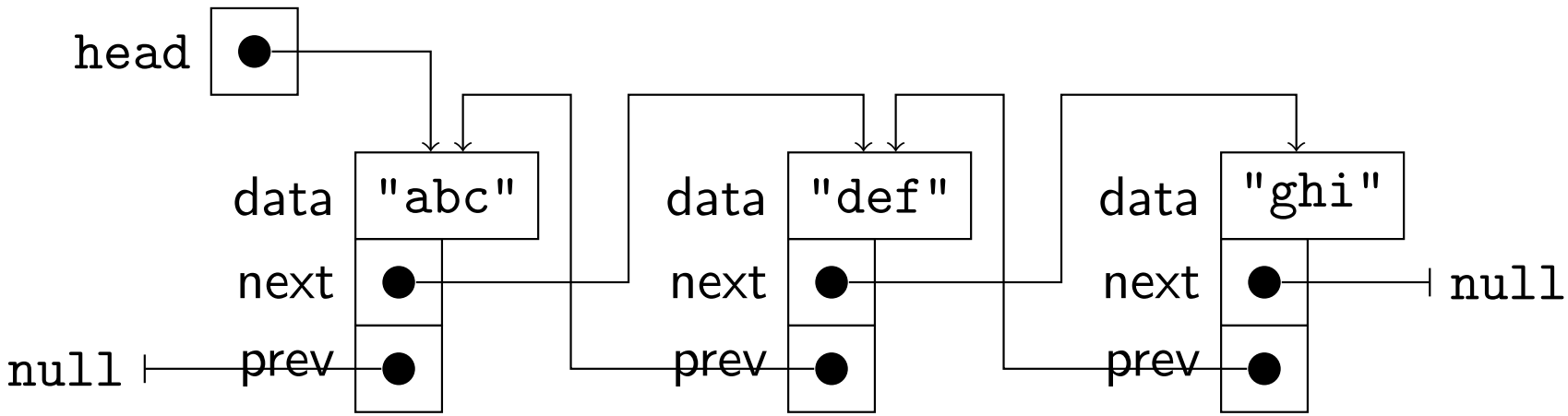
Müllsammlung (garbage collection) läuft.

Doppelt Verkettete Liste

Eine doppelt verkettete Liste speichert zwei Referenzen pro Listeneintrag, einen Vorgänger und einen Nachfolger:

```
class ListEntry {
    Object    data;
    ListEntry prev;
    ListEntry next;
}
```

Das macht Löschen einfacher. Auch kann man die Liste rückwärts durchlaufen.

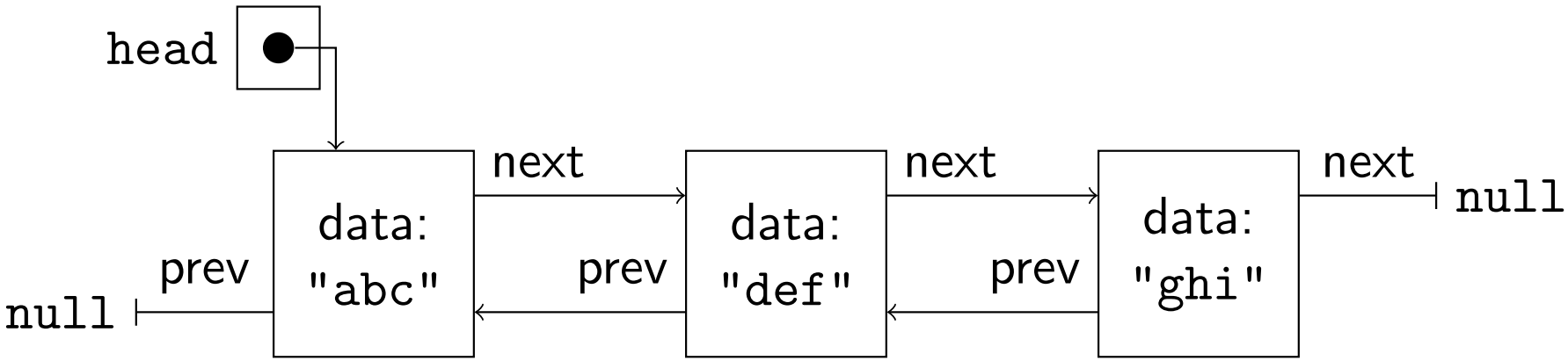


Doppelt Verkettete Liste

Eine doppelt verkettete Liste speichert zwei Referenzen pro Listeneintrag, einen Vorgänger und einen Nachfolger:

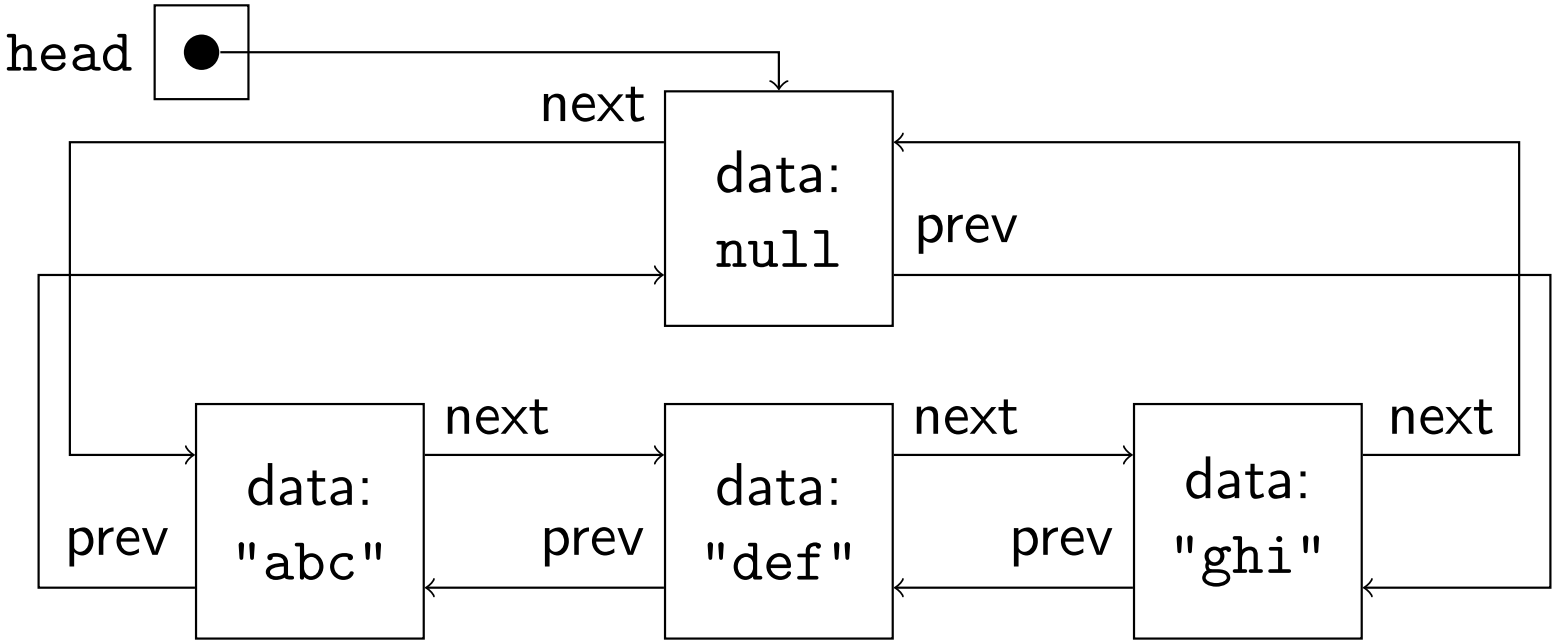
```
class ListEntry {
    Object    data;
    ListEntry prev;
    ListEntry next;
}
```

Das macht Löschen einfacher. Auch kann man die Liste rückwärts durchlaufen.



Blindelement

Statt einer Referenz auf `null` benutzt man bei doppelt verketteten Listen gerne ein Blindelement:



Dadurch kann man sich Fallunterscheidungen sparen. Man spricht hier auch von einer zyklisch verketteten Liste.

Doppelt verkette Liste

Eine Implementierung ist:

```
public class DoublyLinkedList
    ListEntry head;

    /** Erzeugt eine neue leere Liste. */
    public List() {
      head = new ListEntry();
      head.next = head.prev = head;
    }

    /** Prueft, ob die Liste leer ist. */
    public boolean isEmpty() {
      return head.next == head;
    }
    ...
```

Einfügen in doppelt verkettete Listen

Hier fügen wir ein Element hinten in die Liste ein.

```
//public class DoublyLinkedList
    ...
    /** Fuegt ein neues Element ein. */
    public void add(Object data) {
        ListEntry newEntry = new ListEntry();
        newEntry.data = data;
        newEntry.next = head;
        newEntry.prev = head.prev;
        newEntry.prev.next = newEntry;
        newEntry.next.prev = newEntry;
    }
    ...
```

Man kann aber auch vorne einfügen. Dafür muss man nur die zweite und dritte Zeile anpassen.

Iterieren

Beim Iterieren muss man bis zum Blindelement laufen:

```
//public class DoublyLinkedList
...
/** Gibt alle Elemente als String zurueck */
public String toString() {
    String content="";
    String comma="";
    for (ListEntry e = head.next;
         e != head; e = e.next){
        content += comma + e.toString();
        comma = ",";
    }
    return "[" + content + "]";
}
```

Löschen – Teil 1

Auch hier muss ein Element zunächst gefunden werden, um es zu löschen.

```
//public class DoublyLinkedList
...
/** Entfernt das Element data aus der Liste. */
public boolean remove(Object data) {
    ListEntry elem = head.next;
    while (elem != head
           && !elem.data.equals(data)) {
        elem = elem.next;
    }
    ...
```

Ist `elem == head`, so wurde das Element nicht gefunden. Andernfalls zeigt `elem` auf das zu löschende Element.

Löschen – Teil 2

Dank des Blindelements, entfällt im zweiten Teil eine Fallunterscheidung.
Das funktioniert auch mit einelementigen Listen.

```
...//boolean remove(Object data)
    if (elem == head) {
        // data wurde nicht gefunden.
        return false;
    } else {
        // Element aus der Liste loeschen.
        elem.prev.next = elem.next;
        elem.next.prev = elem.prev;
        return true;

    }
}
```

Ausblick: Bibliotheken

Man muss beim Programmieren das Rad nicht immer neu erfinden.

- `java.util.LinkedList` ist eine Implementierung für verkettete Listen in Java.
- Es gibt auch anders implementierte Listen, wie `java.util.ArrayList`. Diese haben Vor- aber auch Nachteile. Um diese einschätzen zu können sollte man wissen, wie verkettete Listen funktionieren.
- In vielen Fällen reicht es aus eine Implementierung aus der Java-Bibliothek zu verwenden. Nur selten braucht man spezielle „handmodellierte“ Datenstrukturen.